Lexers in Programming Languages

Reading

Some optional links:

- Info on OCAML's lex/yacc[1]
- The Lexing module[2]
- Flex source page — C-code lexer[3]
- Stanford Compilers MOOC[4] an excellent free online course to learn the basics of compiler design

Stages in a Compiler

The theory of finite automata and regular expression is the essential ingredient of standard lexical analysis tools, typically called **Lexers**.

Briefly a program when processed by a compiler goes through the following phases:

Lexical Analysis During lexical analysis we take the input text file, and identify the *lexemes*, or tokens, i.e. the individual "words" that form the program. Users typically describe these lexemes via regular expressions, and the **Lexer** is a program that takes this description and produces a DFA that drives this separation of the text in lexemes. The module produced by the lexer can then be included in the other steps.

Syntactic Analysis During syntactic analysis we take this stream of lexemes produced by the lexical analysis phase, and turn these into a more abstract form, typically called the *abstract syntax tree* (AST). This is effectively the step in normal language that takes the words and forms them into sentences. Tools called "parser generators" produce a module that performs this step based on descriptions for context-free-languages that we will see in the future.

Semantic Analysis During semantic analysis we analyze the AST produced by the syntactic analysis step, to determine the "meaning" of the individual parts. This could include for example type-checking.

Intermediate Code Generation Typically a next step is to turn the AST into some intermediate language code, with a small set of instructions. This represention is agnostic to the computer architecture, but is close enough that generating machine code from it is relatively easy.

[1] http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html
[2] http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html
[3] http://flex.sourceforge.net/
[4] https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about

Code Optimization A number of techniques exist to try to optimize the intermediate code, storing often-used computations, removing steps that can be avoided, inlining some function calls and others. In modern compilers, this step constitutes the bulk of the compiler's work.

Code Generation At this stage the intermediate language code is turned into machine instructions and is written into an executable file, or typically assembly code. This step will depend on the specific processor architecture used and what processor instructions are available.

For the time being, we will only concern ourselves with the lexical analysis portion. We will take a glimpse at the syntactic analysis/parsing portion when we cover context-free grammars.

Lexical Analysis and lexers

A lexer expects as input a file that is in a very specific form, which differs from language to language, but in general shares some features. Here is how a sample file might look like for ocamllex (this file resides in ocaml/parsing/lexer.mll):

```
open Lexing

exception Eof

type tokens = INT of int
            | FLOAT of float
            | LPAREN
            | RPAREN
            | PLUS
            | MINUS
            | TIMES
            | DIVIDE

let print_token tk =
   match tk with
      INT i   -> "INT " ^ string_of_int i
    | FLOAT f -> "FLOAT " ^ string_of_float f
    | LPAREN  -> "LPAREN"
    | RPAREN  -> "RPAREN"
    | PLUS    -> "PLUS"
    | MINUS   -> "MINUS"
    | TIMES   -> "TIMES"
    | DIVIDE  -> "DIVIDE"

}

let digit = ['0'-'9']
let int = '-'? digit+
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let float = digit* frac? exp?
let white = [' ' '\t']+
```

```
let newline = '\r' | '\n' | "\r\n"

rule read = parse
   | white { read lexbuf }
   | newline { read lexbuf }
   | int as i { INT (int_of_string i) }
   | float as f { FLOAT (float_of_string f) }
   | '('  { LPAREN }
   | ')'  { RPAREN }
   | '+'  { PLUS }
   | '-'  { MINUS }
   | '/'  { DIVIDE }
   | eof  { raise Eof }
```

These files typically look like OCAML files, but with some specific and different syntax rules.

In this case there is a first section of the file enclosed in braces, called the preamble. We can set up some initial instructions here. For us these are opening the Lexing module so we have easy access to its methods (i.e. we could write foo instead of Lexing.foo some times), and specifying the type of the tokens we want to produce (we can call the type anything we want of course).

I have also included here a method that produces a string representation of the token, for printing purposes. Both this method and the type declaration could have gone into a separate ml or mli file instead, and then loaded in.

Following the preamble is a series of let statements for providing shortcuts of regular expressions. The right-hand-side of these let statements is not normal OCAML code, but instead a regular expression following the description in section 12.2.4 of the manual[5]. You can use these bindings in future let statements.

Following that is a rule, specifying a function, in this case called read, that would receive as input the stream of input. That function does a pattern match based the various regular expressions. In braces we describe what should happen in each case (they should all return the same type, in our case tokens). There are two rules:

- Whichever regular expression matches the longest string of input will apply.
- If there is a tie, two regular expressions matching the same length, the one that appears first applies (so for instance keywords need to go before identifiers in a language compiler).

We "compile" this set of instructions:

```
$ ocamllex lexer.mll
16 states, 354 transitions, table size 1512 bytes
```

And we get a file lexer.ml that tends to not be very readable, but contains instructions for carrying out the produced DFA. As you can see, we get some information about that DFA.

[5] http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html

You can see a short execution of this lexer by looking into the driver1.ml file, and compiling and running it.